REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1997	Final Report

4. TITLE AND SUBTITLE

The mpC Programming Environment for Distributed Memory Machines

5. FUNDING NUMBERS

F6170897W0099

6. AUTHOR(S)

Prof. Victor Ivannikov

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute of Systems Programming of the Russian Academy of Sciences
Bolshaya Communisticheskaya 25
Moscow 109004
Russia

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

SPC 97-4028

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

This report results from a contract tasking Institute of Systems Programming of the Russian Academy of Sciences as follows: The contractor will develop a prototype parallel programming environment that addresses large and fine-grain parallelism, control-parallel and data-parallel paradigms.

14. SUBJECT TERMS

Advanced Computing

15. NUMBER OF PAGES

22

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

pQ

The mpC Programming Environment for Distributed Memory Machines (Final Report on Special Contract SPC-97-4028)

Authors: Alexey L. Lastovetsky (project leader)
Alexey Ya. Kalinov (senior researcher)
Ilya N. Ledovskih (researcher)
Dmitry M. Arapov (researcher)
Mikhail A. Posypkin (junior researcher)

Russian Academy of Sciences
Institute for System Programming
25 Bolshaya Kommunisticheskaya str.
Moscow 109004
Russia

Director: Prof. Victor P. Ivannikov
Phone: 7(095)912-4425
Fax: 7(095)912-1524
E-mail: ivan@ispras.ru

Moscow
October 1997

1. Introduction

Heterogeneous networks of diverse workstations, servers, PCs and parallel computers become most common parallel architectures available. Such networks are characterised by the diversity of performances of computing nodes as well as the diversity of bandwidths and speeds of communication links. Progress in network technologies heightens performance potential of networks of computers (note, that throughout most of the 1990s network technology outstripped processor technology [1]). Therefore, often more performance can be achieved by using the same set of computers utilized via up-to-date network equipment as a single distributed memory machine rather than with a new more powerful computer.

To utilize a heterogeneous network of computers as a single distributed memory machine, dedicated tools are needed. To be useful for a wide range of users, such tools should:

- support efficient parallel programming (to exploit performance potential of particular networks);
- support portable parallel programming (to allow applications once developed for a particular network to run on other networks);
- support modular parallel programming (to allow to develop and use parallel libraries);
- support efficiently portable parallel programming (to allow to develop portable programs adaptable to peculiarities of a particular executing network to exploit its performance potential as efficient as possible);
- support an easy-in-use programming model (to allow the development of really complex and reliable parallel applications).

There are two main approaches to portable programming distributed memory machines. The first one is based on high-level programming languages like HPF [2]. HPF supports an easy-in-use model of parallel programming. It was standardized a couple years ago. High-quality portable HPF compilers (for example, from the Portland Group) compliant to the HPF Standard have already appeared and can be used to develop portable and modular parallel applications. But HPF as well as its predecessor Fortran D [3] and other high-level programming languages (such as Dataparallel C [4], Modula-2* [5], etc.) are not intended for programming heterogeneous network of computers. The point is that homogeneous multiprocessor with very fast communications among their nodes is an inherent model underlying these languages (as well as some packages like ScaLAPACK [6]). The model comes from massively parallel computers being originally a target architecture for these languages. Therefore, HPF, supporting efficiently portable parallel programming for massively parallel computers and homogeneous clusters of workstations connected with very fast network equipment, does not support that for heterogeneous networks.

The second approach ensures efficient parallel programming a particular network of computers and is based on message-passing function extensions of C, C++ and Fortran 77 like MPI (Message Passing Interface).

There are numerous run-time systems to utilize a heterogeneous network of computers as a single distributed memory machine (such as PVM [7], Nexus [8], PARMACS [9], p4 [10] etc.), but currently most research efforts in this area concentrate about MPI. Unlike other popular message-passing package - PVM, MPI has been standardized as MPI 1.1 [11] and widely implemented in compliance with the Standard. Therefore, MPI supports portable parallel programming. The second important advantage of MPI is that the notion of communicator introduced in MPI allows to write and compile independently different modules of the entire parallel program. That is, MPI (unlike, say, PVM) supports modular parallel programming and, hence, the development of parallel libraries.

A big group of researchers continue working on MPI functionalities to prepare the MPI-2 standard [12]. Planned additions include dynamic processes, one-sided communications, extended collective operations, external interfaces, additional language bindings, and I/O.

Now the main research problem, related to MPI, concludes in such implementations of MPI that run on a wide range of essentially heterogeneous networks and provide efficient communications. Portable implementations of MPI (such as MPICH), implementation of broadcasting via multicasting, mulitiprotocol implementations of MPI permitting different communication methods to coexist (such as the Nexus-based MPICH [13]) and the development of covering tools to communicate among processes running under different uniprotocol MPI implementations (such as PVMPI [14] using PVM to do it) are typical up-to-date research efforts in this direction. They lead to more efficient and portable MPI implementations.

At the same time, MPI has two disadvantages. The first one is a low level of its parallel primitives. It is tedious and error-prone to write in MPI really complex and useful parallel applications. Note, that this disadvantage is not due to the MPI design (which is really excellent), but due to the message-passing paradigm.

The second disadvantage is that MPI (both MPI 1.1 and upcoming MPI-2) does not support efficient portability. Efficient portability means that a parallel application, running efficiently on a particular heterogeneous network, will run efficiently after porting to other heterogeneous network. In other words, an efficiently portable application can adapt to peculiarities of underlying heterogeneous network (the number and performances of processors, bandwidths and speeds of links among them). Of course, because of low level of MPI, one may write a special porting system to provide efficient portability of his application, but such a system is usually too complicated, and the necessity of its development can frighten off most of normal users.

Another research direction in run-time systems supporting programming for heterogeneous networks addresses load balancing. There are a number of researchers working on run-time systems supporting migration of processes and/or data among processor nodes of executing heterogeneous network to balance their loads [15-16]. Such systems monitor loads of processor nodes and redistribute resources if the loads are not balanced. Although such systems allow to adapt running parallel applications to underlying heterogeneous networks, they do not solve the problem of efficient portability. The point is that load balancing cannot provide the best execution time of a parallel applications, which depends on the number of processes, the volume of computations to perform by each of the processes, initial distribution of processes over the heterogeneous network, monitoring overheads and so on. Since such a system knows nothing specific about a running application and is hidden from the user (who even may not know about it), the system will provide the best execution time only under very strong restrictions on the application and the underlying hardware.

So, traditional tools do not support efficiently portable modular parallel programming heterogeneous networks of computers.

2. Outline of main results

We have designed a programming language named mpC and aimed at efficiently portable modular parallel programming.

A prototype mpC programming environment has been designed and implemented. In October 1996, an alpha version of the environment was put for public access in Internet at the address http://www.ispras.ru/~mpc. Currently, version 1.2.1 of the mpC system is released. It runs on networks of diverse workstations and PCs running Solaris, SunOS 4.1.3, HP-UX, Linux, OSF1 and using LAM MPI [17] or MPICH [18] as a communication platform. There are more than 200 mpC users around the world and more than 2300 visits at the mpC page since February 1997.

The mpC language and its supportive environment have been used or plan to be used as a basis for courses on parallel and distributed computing in several universities in Russia and USA.

Basic techniques of the writing of efficiently-portable mpC applications have been developed. A number of mpC applications solving some typical problems have been written and compared to their counterparts developed with traditional tools (MPI, HPF, ScaLAPACK, etc.). Numerous experiments showed that the mpC applications run on heterogeneous networks of computers essentially faster than their traditional counterparts.

A number of papers on mpC has been published [19-23]. Additionally, detailed documentation on mpC (including language specification, installation guide, user's guide, related publications, sample mpC applications, etc.) as well as the corresponding free software are available at the mpC homepage (http://www.ispras.ru/~mpc).

By now, mpC remains a unique tool (we do not know other commercial or research programming environments supporting efficient portability).

Currently, the mpC programming environment includes a compiler, a run-time support system, a library, and a command-line user interface. The compiler translates a source mpC program into ANSI C code with calls to functions of the run-time support system. The run-time support system manages the computing space which consists of a number of processes running over underlying distributed memory machine as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a subset of MPI). It ensures platform-independence of the rest of system components. The library consists of a number of functions which support debugging mpC programs as well as provide some low-level efficient facilities. The command-

line user interface consists of a number of commands supporting the creation of a virtual distributed memory machine and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector running a special benchmark and saved in a file used by the run-time support system.

The mpC language is an ANSI C superset allowing the user to specify topology of and to define so-called network objects (in particular, in run time) as well as to distribute data and computations over the network objects. The mpC programming environment uses this information to map (dynamically) the mpC network objects to any underlying heterogeneous network in such a way to ensure the most efficient running of the application on the network. For example, the user can write in mpC the following

```
/*1*/  nettype HeteroNet(n, p[n]) {
/*2*/    coord I=n;
/*3*/    node { I>=0: p[I]; };
/*4*/  };
/*5*/  ...
/*6*/     {
/*7*/       repl int m, q[N];
/*8*/       ... /* Computing m and q[0],...,q[m-1] */
/*9*/       {
/*10*/        net HeteroNet(m,q) r;
/*11*/        ...
/*12*/      }
/*13*/    }
```

to define automatic network object `r` consisting of `m` virtual processors, the relative performance of the `i`-th virtual processor being characterized by the value of `q[i]`.

Here, lines 1-4 declare network topology `HeteroNet` parametrized with integer parameter `n` and vector parameter `p` consisting of n integers. Line 2 is a coordinate declaration declaring the coordinate system to which virtual processors are related. It introduces coordinate variable `I` ranging from `0` to `n-1`. Line 3 is a node declaration. It relates virtual processors to the coordinate system and declares their types and performances. It stands for the predicate *for all `I<n` if `I>=0` then a virtual processor, whose relative performance is specified by the value of `p[I]`, is related to the point with the coordinate `[I]`.*

Line 7 defines variable `m` and array `q` both replicated over the entire computing space (any network object is a region of the entire computing space). By definition, data object distributed over a region of the computing space comprises a set of components of any one type so that each virtual processor of the region holds one component. By definition, a distributed data object is replicated if all its components is equal to each other.

Conceptually, creation of a new network object is initiated by a virtual processor of a network object already created. This processor is called a parent of the created network object. The parent belongs to the created network object. In our case, the parent of network object `r` is the so-*called virtual host-processor* - the only virtual processor defined from the beginning of program execution till program termination.

Suppose we to model the evolution of `m` groups of bodies under the influence of Newtonian gravitational attraction, and our parallel application uses a virtual processor to update a single group. Suppose also `q[i]` to be equal to the square of the number of bodies in the `i`-th group. Then, line 10 defines network object `r`, executing most of computations and communications, in such a way, that it consists of `m` virtual processors, and the relative performance of each processor is characterized by the volume of computations to update the group which it computes. So, the more powerful is the virtual processor, the larger group of bodies it computes. The mpC programming environment bases on this information as well as on the information about the topology of the underlying heterogeneous network to map the virtual processors into the processes, running on this heterogeneous network and representing the entire computing space, in the most appropriate way. Since it does it in run time, the user does not need to recompile the mpC code to port it to another heterogeneous network.

Additionally, to write an application more adaptable to the underlying hardware, one can use calls to the following library functions:

```
int MPC_Processors_static_info(int* num_of_processors, double** performances);
int MPC_Links_static_info(MPC_Links ** links);
```

A call to the first function returns the number of actual processors and their relative performances. A call to the second one returns information about the network structure and bandwidths and speeds of

communication links. Suppose we to multiply 2 dense square kxk matrices X and Y, and our parallel application uses a number of virtual processors, each of which computes a number of rows of the resulting matrix Z. Finally, let we use in line 8 the following code

```
repl double *powers;\
repl MPC_Links *links;\
external repl k;\
MPC_Processors_static_info(&m,&powers);\
MPC_Links_static_info(&links);\
Partition(&m, powers, links, q, k);
```

to compute m, `q[0]`,...,`q[m-1]`. Based on the number and the performances of actual processors as well as network characteristics, the user-defined function `Partition` computes how many actual processors will take part in multiplying the matrices and how many rows of the resulting matrix will be computed by each of these actual processors (note, that the number of actual processors participating in the matrix multiplication may be less than the total number of actual processors, because communication overheads may exceed parallelization speedup). So, after the call to this function m will hold the number of participating actual processors and `q[i]` will hold the number of rows computed by `i`-th actual processor. Network object `r`, which executes the rest of computations and communications, is defined in such a way, that the more powerful the virtual processor, the greater number of rows it computes. The mpC environment will ensure the optimal mapping of the virtual processors constituting `r` into a set of processes constituting the entire computing space. So, not more than one process from processes running on each of actual processors will be involved in multiplication of matrices, and the more powerful the actual processor, the greater number of rows its process will compute.

3. Latest achievements

3.1. Language

A simplified form of node declarator was introduced. Note, that the node declarator

```
node { I>=0: p[I]; };
```

used in the above mpC fragment is the simplified form of the node declarator

```
node { I>=0: fast*p[I] scalar; };
```

A relation declaration to specify relations between subnetworks of the same network was introduced. The partial order "to be a subnetwork of" is defined on a set of subnetworks of the same network. The compiler needs the relation for correct translation of many expressions. Therefore, the partial order should be explicitly specified in mpC program. There was only one way to do it. Namely, the relation could be specified with a subnetwork declaration, similar to the declaration of subnetwork sr3 in the following fragment

```
repl [*]a[6]={10, 20, 30, 40, 50, 60};
net HeteroNet(6, a) r6;
subnet [r6:I<5] sr5;
subnet [sr5:I<3] sr3;
subnet [r6:I<4] sr4;
```

The declaration specifies that `sr3` is a subnetwork of `sr5`. At the same time, although, in fact, `sr4` is a subnetwork of `sr5`, the compiler will treat them as incomparable ones. Now, to inform the compiler about this fact, one can use the relation declaration

```
relation sr4<sr5;
```

in the corresponding block.

3.2. Run-time support system

The run-time support system was ported to SunOS 4.1.3, OSF1 and Windows 95. In addition to LAM MPI and MPICH, it was tested with HP MPI - an implementation of MPI from Hewlett-Packard.

Algorithms of managing processes were optimized to lower communication overheads.

The model of a heterogeneous network of computers and the algorithm of load balancing were improved.

More debugging facilities were added.

A number of new functions to get the information about executing environment were provided. In particular, they include the function `MPC_Total_nodes` returning the total number of virtual processors constituting the computing space as well as the function `MPC_Links_static_info` returning information about the network structure and bandwidths and speeds of communication links of the underlying hardware.

3.3. Compiler

The compiler was ported to SunOS 4.1.3, OSF1 and Windows 95.

Now, the compiler supports old-style function definitions.

Now, the compiler supports an incremental mode of compiling.

Algorithms of translating network definitions were simplified to improve reliability as well as optimized to lower communication overheads (see Appendix A for more details).

3.4. Command-line user interface

The command-line user interface was simplified. It was reimplemented to run on top of both LAM MPI and MPICH (see Appendix B for more details).

In addition to Solaris, HP-UX and Linux, it was ported to SunOS 4.1.3 and OSF1.

3.5. Applications`

Some efficiently-portable mpC applications solving linear algebra problems were developed, implemented and compared to ScaLAPACK counterparts (see Appendix C for more details about parallel Cholesky factorization).

Some recommendations on the writing of efficiently-portable mpC applications solving irregular and regular problems were formulated.

An irregular problem is characterized by some inherent coarse-grained or large-grained structure implying quite deterministic decomposition of the whole program into a set of programs running in parallel and interacting via message passing. As rule, there are essential differences in volumes of computations and communications to perform by different programs. So, in mpC one can define a network object, to execute the set of programs, and specify performances of its virtual processors and lengths of its links in accordance with the different volumes of computations and communications, and mpC programming environment will map the virtual processors onto underlying hardware in such a way to ensure an efficient execution of the application.

Unlike an irregular problem, for a regular problem a decomposition of the whole program into a large set of small equivalent programs, running in parallel and interacting via message passing, is the most natural one. The main idea of efficient solving a regular problem is to reduce it to an irregular problem the structure of which is determined by the topology of underlying hardware rather than the topology of the problem. So, the whole program is decomposed into a set of programs each made of a number of the small equivalent programs stuck together and running on a separate processor of the underlying hardware. To do it in mpC, one can detect with a special library function the topology of the underlying hardware, define a network object having the same topology and distribute computations and communications in accordance with the topology to ensure efficient execution of the program.

4. References

[1] H.El-Rewini, and T.Lewis, Introduction To Distributed Computing. Manning Publications Co., 1997 (to appear).

[2] High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1. Rice University, Houston TX, November 10, 1994

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, Fortran D Language Specification. Center for Research on Parallel Computation, Rice University, Houston, TX, October 1993.

[4] P. J. Hatcher, and M. J. Quinn, Data-Parallel Programming on MIMD Computers. The MIT Press, Cambridge, MA, 1991.

[5] M. Philippsen, and W. Tichy, "Modula-2* and its compilation", First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991.

[6] J.Choi, J. Demmel, I.Dhillon, J.Dongarra, S.Ostrouchov, A.Petitet, K.Stanley, D.Walker, and R.Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance, UT, CS-95-283, March 1995.
[7] A.Geist, A.Beguelin, J.Dongarra, W.Jlang, R.Manchek,V.Sunderam, PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.
[8] The Nexus Multithreaded Runtime System, http://www.mcs.anl.gov/nexus/index.html.
[9] R.Calkin, R.Hempel, H.-C.Hoppe, and P.Wypior, "Portable Programming with the Parmacs message-passing library", Parallel Computing, Special issue on message-passing interface (to appear).
[10] R.Butler and E.Lusk, "Monitors, Messages, and Clusters: the p4 Parallel Programming System", Journal of Parallel Computing, 20:547-568, 1994.
[11] Message Passing Interface Forum, MPI: A Message-passing Interface Standard, version 1.1, June 1995.
[12] MPI-2: Extensions to the Message-Passing Interface, http://www.mcs.anl.gov/mpi/.
[13] I.Foster, J.Geisler, C.Kesselman, and S.Tuecke, "Managing Multiple Communication Methods in High-Performance Networked Computing Systems", Journal of Parallel and Distributed Computing (to appear).
[14] G.Fagg, and J.Dongarra, "PVMPI: Inter-operation and Co-ordination of MPI-1 Applications across multiple systems and MPI implementations", http://www.cs.utk.edu/~fagg/mpiglue/.
[15] Dome: Distributed Object Migration Environment, http://www.cs.cmu.edu/~Dome/.
[16] Hector: A Heterogeneous Task Allocator, http://www.erc.msstate.edu/~russ/hpcc/.
[17] LAM/MPI Parallel Computing, http://www.osc.edu/lam.html.
[18] MPICH -A Portable Implementation of MPI, http://www.mcs.anl.gov/mpi/mpich/.
[19] A.Lastovetsky, "mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers", ACM SIGPLAN Notices, 31(2), February 1996, pp.13-20.
[20] D.Arapov, A.Kalinov, and A.Lastovetsky, "Managing the Computing Space in the mpC Compiler", Proceedings of the 1996 Parallel Architectures and Compilation Techniques (PACT'96) conference, IEEE CS Press, Boston, MA, Oct. 1996, pp.150-155.
[21] D.Arapov, A.Kalinov, and A.Lastovetsky, "Resource Management in the mpC Programming Environment", Proceedings of the 30th Hawaii International Conference on System Sciences (HICSS'30), IEEE CS Press, Maui, HI, January 1997.
[22] D. Arapov, V.Ivannikov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis, "Modular Parallel Programming in mpC for Distributed Memory Machines", Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs'97), IEEE Computer Society Press, Aizu-Wakamatsu, Japan, March 1997, pp.248-255.
[23] D.Arapov, A.Kalinov, A.Lastovetsky, I.Ledovskih, and T.Lewis, "A Programming Environment for Heterogeneous Distributed Memory Machines", Proceedings of the 1997 Heterogeneous Computing Workshop (HCW'97) of the 11th International Parallel Processing Symposium (IPPS'97), IEEE CS Press, Geneva, Switzerland, April 1997, pp.32-45.

Appendix A

A.1. Translation of network definitions

The mpC compiler translates a source mpC file into a target ANSI C file with calls to functions of the run-time support system. It uses the SPMD model of target code, when all processes constituting the target mpC program run identical code.

All processes constituting the target program are divided into 2 groups - a special process, called *dispatcher*, playing the role of the manager of the computing space, and common processes, called *nodes*, playing the role of virtual processors of the computing space. The dispatcher works as a server. It receives requests from nodes and sends them commands.

In the target program, every network of the source mpC program is represented by a set of nodes called *region*. At any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them are responsibility of the dispatcher. The only exception is the pre-hired *host-node* representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

The main problem in managing processes is hiring them to network regions and dismissing them. A solution of this problem establishes the whole structure of the target code and forms requirements for functions of the run-time support system.

To create a network region, its parent node computes, if necessary, parameters of the corresponding network topology and sends a creation request to the dispatcher. The request contains full topological information about the created region including the number of nodes and their relative performances. On the other hand, the dispatcher keeps information about the topology of the target network of computers including the number of actual processors, their relative performances and the mapping of nodes onto the actual processors. Based on the topological information, the dispatcher selects the set of free nodes, which is most appropriate to be hired in the created network region. (More detailed description how dispatcher does it, may be found in [8].) After that, it sends to every free node a message saying whether the node is hired in the created region or not.

To deallocate a network region, its parent node sends a message to the dispatcher. Note, that the parent node leaves hired in the parent-network region of the deallocated region. The rest of members of the deallocated network region become free and begin waiting for commands from the dispatcher.

Any node can detect its hired/free status. It is hired if a call to function MPC_Is_busy returns 1. If such a call returns 0, the node is free.

Any node can detect if it is hired in some particular region or not. A region is accessed via its descriptor. If the descriptor `rd` corresponds to the region, then a node belongs to the region if and only if the function call `MPC_Is_member(&rd)` returns `1`. In this case, descriptor `rd` allows the node to obtain comprehensive information about the region as well as identify itself in the region. The region descriptor has type `MPC_Net` and holds the following data:

- topological data associated with the region, such as the number of coordinates, an integer array containing actual topological arguments (if any) and the number of elements in this array, pointers to the corresponding topological functions;
- the number of nodes in the region;
- the linear number of the node in the region;
- an integer array containing coordinates of the given node in the corresponding network;
- some additional and /or redundant information aimed at optimization of computations and communications.

When a free node is hired in a network region, the dispatcher must let the node know, in which region it is hired, that is, must specify the descriptor of that region. The simplest way - to pass the pointer to the region descriptor from the parent node through the dispatcher to the free node, is senseless for distributed memory systems not having common address space. Therefore, in addition to the region descriptor, something else is needed to identify the created region in a unique fashion. The additional identifier must have the same value on both the parent and the free node and be passable from the parent node through the dispatcher to the free node.

In a source mpC program, a network is denoted by its name, being an ordinary identifier and not having to have file scope. Therefore, a network name can not serve as a unique network identifier even within a file. One could enumerate all networks declared in the file and use the number of a network as an identifier unique within the file. However, such an identifier being unique within a file can not be used as a unique identifier within the whole program that may consist of several files. Nevertheless, one can use it without collisions when creating network regions, if during network-region creation all participating nodes execute the target code located in the same file. Our compiler just enumerates networks defined in a file and uses their numbers as network identifiers in target code when creating the corresponding network regions. It does ensure that during the creation of a network region all involved nodes execute the target code located in the same file.

Creating a network region involves its parent node, all free nodes and the dispatcher. The parent node calls to function `MPC_Net_Create` declared in the header file mpC.h as follows:

```
int MPC_Net_create(MPC_Name name, MPC_Net* net);
```

where `name` contains the unique number of the created network in the file, and `net` points to the corresponding region descriptor. The function computes all topological information and sends a creation request to the dispatcher.

Meantime, free nodes are waiting for commands from the dispatcher at so-called waiting point calling the function `MPC_Offer` declared in mpC.h as follows:

```
int MPC_Offer(const MPC_Name* names, MPC_Net** nets_voted, int voted_count);
```

where `names` is an array of numbers of all networks the creation of which are expected at the waiting point, `nets_voted` points to an array of pointers to descriptors of the regions the creation of which are expected at the waiting point, `voted_count` contains the number of elements in array names.

The correspondence between the network numbers and region descriptors is established in the following way. If a free node receives from the dispatcher a message saying that it is hired in a network the number of which is equal to `names[i]`, then the node is hired in the network region the descriptor of which is pointed by `nets_voted[i]`.

A free node leaves the waiting function `MPC_Offer` either after it becomes hired in a network region or after the dispatcher sends to all free nodes the command to leave the current waiting point.

A.2. Structure of target code for mpC block

In general, target code for an mpC block with network definitions has two waiting points. In the first, called *creating waiting point*, free nodes are waiting for commands on region creations. In the second, called *deallocating waiting point*, they are waiting for commands on region deallocations. In general, free nodes participate not only in creation/deallocation of regions for networks defined in the mpC block, but also in overall computations (that is, in computations distributed over the entire computing space) and/or in creation/deallocation of regions for networks defined in nested blocks. Let us call the first mpC statement in the block involving all free nodes in its execution a waiting-point break statement.

Then, in the most general case, the compiler generates target code of the following structure:

```
{
  declarations
  {
   if(!MPC_Is_busy()) {
    target code executed by free nodes to create regions for
    networks defined in source mpC block
   }
   if(MPC_Is_busy()) {
    target code executed by hired nodes to create regions for
    networks defined in source mpC block
    and
    target code for mpC statements before waiting-point break statement
   }
   epilogue of waiting point
  }
  target code for mpC statements starting from waiting-point break statement
  {
   target code executed by hired nodes to deallocate regions for
   networks defined in source mpC block

    label of deallocating waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes to deallocate regions
      for networks defined in source mpC block
    }
    epilogue of waiting point
  }
}
```

If the source mpC block does not contain a waiting-point break statement (that is, overall statements and nested blocks with network definitions or overall statements), then creating and deallocating waiting points can be merged. Let us call such a waiting point *shared waiting point*. Target code for the mpC block with a shared waiting point looks as follows:

```
{
  declarations
  {
    label of shared waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes to create and deallocate
      regions for networks defined in source mpC block
    }
```

```
    if(MPC_Is_busy()) {
      target code executed by hired nodes to create and deallocate
      regions for networks defined in source mpC block
      and
      target code for statements of source mpC block
    }
    epilogue of waiting point
  }
}
```

To ensure that during the creation of a network region all involved nodes execute target code located in the same file, the compiler put a global barrier into the epilogue of waiting point.

The coordinated arrival of nodes to the epilogue of waiting point is ensured by the following scenario:

- the host makes sure that all other hired nodes, which might send a creation/deallocation request expected in the waiting point, have already reached the epilogue;
- after that, the host sends a message, saying that any creation/deallocation request expected in the waiting point will not appear yet, to the dispatcher;
- after receiving the message the dispatcher sends all free nodes a command ordering to leave the waiting point;
- after receiving the command each free node leaves the waiting function and reach the epilogue.

A.3. Process management in details

To introduce the process management in more details, let us consider the following mpC file:

```
/*1 */  nettype T(m) { coord I=m; };
/*2 */  void [*]f(int [host] hn) {
/*3 */    net T(2) n;
/*4 */    repl in;
/*5 */    in=hn;
/*6 */    {
/*7 */      net T(in) [n] nn;
/*8 */      .../* declarations*/
/*9 */      .../*statements without a waiting-point break statement*/
/*10*/    }
/*11*/  }
```

Line 1 introduces topology `T` with parameter `m`. It describes networks consisting of `m` virtual processors with the integer coordinate variable `I` ranging from `0` to `m-1`.

Line 3 defines network `n` consisting of two virtual processors.

Line 4 defines integer variable `in` replicated over the entire computing space.

Line 5 broadcasts the value of variable `hn` from the virtual host-processor over the entire computing space. The statement is executed by the entire computing space. Therefore, it is a waiting-point break statement for the function body.

Line 7 defines network `nn`. The network `nn` is a distributed network. In general, mpC allows to define not only a single network but also a set of single networks by means of defining so-called distributed network. A definition of a distributed network specifies the type of the network and its parent network. Such a definition may be considered as a distributed over the parent network definition of a single network of the specified type. The parent network of a distributed network can also be distributed. But in any case, a distributed network is a set of single networks of the same type. The number of single networks in this set is equal to the number of virtual processors in the parent network each of the virtual processors of the parent network being a parent of a single network of the set.

There are not facilities to specify a single network belonging to a distributed network in mpC. Therefore, whenever one specifies a subnetwork of a distributed network, he means a set of subnetworks of the single networks constituting the distributed network. Similarly, if one specifies a single processor of a distributed network, he means a set of single processors of the single networks constituting the distributed network. Any computation on a distributed network is divided into independent computations on the single networks constituting the distributed network.

So, network `nn` distributed over its parent network `n` divides into a set of two single networks the type of which is defined completely only in run time.

There will be all three kinds of waiting points in target code for function `f`. The function body, where network `n` is defined, contains a waiting-point break statement. Therefore, target code for the function body will contain both creating and deallocating waiting points. The nested block (lines 6-12),

where network nn is defined, does not contain a waiting-point break statement. Therefore, target code for the nested block will contain a shared waiting point.

The following target code

```
/*1 */  void f() {
/*2 */    int MPC_Net_n_6_coord[1];
/*3 */    MPC_Parameters MPC_Net_n_6_params[1]={2};
/*4 */    MPC_Net MPC_Net_n_6={.../* initialization list */};
/*5 */    int in;
/*6 */    {
/*7 */      if(!MPC_Is_busy()) {
/*8 */        MPC_Name MPC_names[1]={6};
/*9 */        MPC_Net* MPC_nets[1];
/*10*/        MPC_nets[0]=&MPC_Net_n_6;
/*11*/        MPC_Offer(MPC_names, MPC_nets,1);
/*12*/      }
/*13*/      if(MPC_Is_busy()) {
/*14*/        if(MPC_Is_member(&MPC_Net_host)) {
/*15*/          MPC_Net_create(6, &MPC_Net_n_6);
/*16*/        }
/*17*/        if(MPC_Is_host()) {
/*18*/          MPC_Host_out();
/*19*/        }
/*20*/      }
/*21*/      MPC_Waiting_point_end();
/*22*/    }
/*23*/    /* target code for in=ih */
/*24*/    {
/*25*/      int MPC_Net_nn_7_coord[1];
/*26*/      MPC_Parameters MPC_Net_nn_7_params[1];
/*27*/      MPC_Net MPC_Net_nn_7={.../*initialization list*/};
/*28*/      /*target code for  declarations of the source nested block*/
/*29*/      {
/*30*/      MPC_waiting_point_2:
/*31*/        if(!MPC_Is_busy()) {
/*32*/          MPC_Name MPC_names[1]={7};
/*33*/          MPC_Net* MPC_nets[1];
/*34*/          MPC_nets[0]=&MPC_Net_nn_7;
/*35*/          MPC_Offer(MPC_names,MPC_nets,1);
/*36*/        }
/*37*/        if(MPC_Is_busy()) {
/*38*/          if(MPC_Is_member(&MPC_Net_n_6)) {
/*39*/            MPC_Net_nn_7.count=1;
/*40*/            MPC_Net_nn_7.params=MPC_Net_nn_7_params;
/*41*/            {
/*42*/             MPC_Net_nn_7_params[0]=in;
/*43*/            }
/*44*/            MPC_Net_create(7,&MPC_Net_nn_7);
/*45*/          }
/*46*/          /*target code for statements of the source nested block*/
/*47*/          if(MPC_Is_member(&MPC_Net_nn_7)) {
/*48*/            MPC_Net_free(&MPC_Net_nn_7);
/*49*/            if(!MPC_Is_busy())
/*50*/              goto MPC_waiting_point_2;
/*51*/          }
/*52*/          if(MPC_Is_member(&MPC_Net_n_6)) {
/*53*/            MPC_Local_barrier(&MPC_Net_n_6);
/*54*/          }
/*55*/          if(MPC_Is_host()) {
/*56*/            MPC_Host_out();
/*57*/          }
/*58*/        }
/*59*/        MPC_Waiting_point_end();
/*60*/      }
/*61*/      if(MPC_Is_member(&MPC_Net_n_6)) {
/*62*/        MPC_Net_free(&MPC_Net_n_6);
/*63*/        if(!MPC_Is_busy())
/*64*/          goto MPC_reconfig_point_1;
/*65*/      }
/*66*/      if(MPC_Is_host()) {
/*67*/        MPC_Host_out();
/*68*/      }
```

```
/*69*/    MPC_reconfig_point_1:
/*70*/      if(!MPC_Is_busy()) {
/*71*/        MPC_Offer(NULL,NULL,0);
/*72*/      }
/*73*/      MPC_Waiting_point_end();
/*74*/    }
/*75*/  }
```

is generated by the mpC compiler for the above mpC function `f`. Lines 1-23 and lines 61-75 are generated for the function body, and lines 24-51 are generated for the nested block.

Lines 2-22 of the target code are related to the creating waiting point. The network `n` has obtained number 6 as an identifier unique in the file, and the corresponding network region is accessible via descriptor `MPC_Net_n_6` (line 4). Line 2 defines the one-element `array MPC_Net_n_6_coord` to hold the coordinates of nodes of the region. Line 3 defines and initializes the one-element array `MPC_Net_n_6_params` in such a way that its only element holds integer value 2 as an argument of topology `T` establishing the type of network `n` (namely, the network type `T(2)`). Line 4 defines the region descriptor `MPC_Net_n_6` and initializes all such its members, values of which can be computed in compile time.

The target code in lines 8-11 is executed by all free nodes to create the region represented the network n. Line 8 defines and initializes the one-element array `MPC_names` containing the number of the network the creation of which is expected at the first waiting point. Line 9 defines the one-element array `MPC_nets` to hold a pointer to the region descriptor the creation of which is expected at the first waiting point, and line 10 assigns the proper value to its only element. Line 11 calls to the waiting function `MPC_Offer`. A free node leaves the function either after it becomes hired in region `MPC_Net_n_6`, or after the dispatcher sends to all free nodes the command to leave this waiting point.

The target code in lines 14-19 is executed by all hired nodes to create the region for network n and to reach coordinately the epilogue of the creating waiting point.

Lines 14-16 call to function `MPC_Net_create` on the host to form the corresponding creation request and to send it to the dispatcher. The host is accessible via descriptor `MPC_Net_host`. Any node is detect itself as the host if the function call `MPC_Is_member(&MPC_Net_host)` or the function call `MPC_Is_host()` return `1` on the node.

Lines 17-19 call to function `MPC_Host_out` on the host to send the dispatcher a message saying that all free nodes must leave the waiting point. Since in our example the host is the only node, that can send a creation request expected in the waiting point, it knows that all creation requests expected in the waiting point have already been sent, and it may send the message to the dispatcher.

The statement in line 21 calls to function `MPC_Waiting_point_end`. It is an epilogue of the first waiting point. The call provides a global barrier synchronization and does not let any node to continue until all nodes constituting the entire computing space reach it.

The target code for the nested block (lines 24-60) is related to the shared waiting point. The network nn has obtained number 7 as an unique identifier in the file, and the corresponding network region is accessible via descriptor `MPC_Net_nn_7` (line 27).

Lines 25 defines one-element array `MPC_Net_nn_7_coord` to hold the coordinates of nodes of the region. Line 26 defines the one-element array `MPC_Net_nn_7_params` to hold an argument of topology `T` establishing the type of network nn. Line 27 defines the region descriptor `MPC_Net_nn_7` and initializes all such its members, values of which can be computed in compile time.

The target code in lines 32-35 is similar to the target code in lines 8-11 and executed by all free nodes to create the region represented the network nn.

The target code in lines 38-58 is executed by all hired nodes to create and deallocate the region for network nn, to execute statements of the source mpC nested block, and to reach coordinately the epilogue of the shared waiting point.

Lines 39-44 are executed by two nodes constituting region `MPC_Net_nn_6` in parallel to create two regions representing the distributed network nn. Lines 39-43 compute some attributes of these regions, allowing to establish the type of network nn, and store them in the corresponding members of the region descriptor `MPC_Net_nn_7`.

Line 44 calls to function `MPC_Net_create` to form two corresponding creation requests and to send them to the dispatcher.

Lines 48-50 are executed on the regions representing network nn to deallocate them. Line 48 calls to function `MPC_Net_free`. The function provides a local barrier synchronization over the deallocated regions. After all nodes constituting these regions reach the local barrier, each of two nodes constituting their parent region (that is, region `MPC_Net_nn_6`) send a message to the dispatcher. These two nodes remain to be hired in region `MPC_Net_nn_6`. Meantime other members of the distributed network region become free and jump to the label `MPC_waiting_point_2` of the shared waiting point (line 50). They begin executing the free-node code (lines 32-35) and, eventually, join other free nodes calling the waiting function in line 35.

Lines 52-57 ensure that all nodes reach the epilogue of the shared waiting point coordinately. Since the host is not the only node that can send a request expected in the waiting point, it can not pass over the local barrier in line 53 and call to function `MPC_Host_out` in line 56 to send the dispatcher a message saying that all free nodes must leave the waiting point, until all other nodes able to send a creation/deallocation request reach the local barrier.

As a result, all nodes call to the epilogue function `MPC_Waiting_point_end` (line 59) coordinately.

The rest of the target code generated for the function body (lines 61-74) is related to the deallocating waiting point.

Lines 61-68 are executed by hired nodes to deallocate the region representing the network n and to ensure that all nodes reach the epilogue of the deallocating waiting point coordinately. The node, that becomes free after the call to function `MPC_Net_free` in line 62, jumps to the label `MPC_reconfig_point_1` of the deallocation waiting point (line 64) and calls to the waiting function (line 71). Since the host is the only node that can send a deallocation request expected in the waiting point, it does not need to synchronize its work with some other hired nodes and can call to the function `MPC_Host_out` to send the dispatcher a message saying that free nodes must leave the deallocation waiting point.

Lines 70-72 ensure that all free nodes will receive in time the command from the dispatcher to leave the waiting point and will reach the epilogue function (line 73) coordinately with the hired nodes.

Appendix B

B.1. Definition of terms

The following terms are used in this document:

- *Computing space* (a language term) - a set of typed virtual processors of different performances connected with links of different bandwidths;
- *Distributed memory machine* (an implementation term) - any computing system running MPI (for example, cluster of workstations, heterogeneous network of workstations and/or PCs, a specialized parallel computer, or everything taken together);
- *Virtual parallel machine* (an implementation term) - a set of processes representing virtual processors of the computing space and running over distributed memory machine.
- *Host workstation* (an implementation term) - any workstation or PC that may be used as a working place of the user. It is intended that the user may start running mpC applications only from host workstations.

B.2. Outline of the mpC programming environment

Currently, the mpC programming environment includes a compiler, a run-time support system (RTS), a library, and a command-line user interface. All these components are written in ANSI C.

The compiler translates an mpC program into a ANSI C program with calls to functions of RTS. The compilation unit is a source mpC file. The compiler uses optionally either the SPMD model of target code, when all processes constituting a target message-passing program run the identical code, or a quasi-SPMD model, when it translates the source mpC file into 2 distinct target files: the first for the virtual host-processor and the second for the rest of virtual processors.

RTS manages the computing space and provides any necessary communications. RTS encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of compiler components.

The library consists of a number of functions which provide low-level efficient facilities as well as support debugging mpC programs.

The user interface consists of a number of programs supporting the creation of a virtual parallel machine and monitoring the execution of mpC applications on the machine. While creating the machine, its topology is detected by a topology detector and saved in a file used by RTS. The topology detector executes a special benchmark to detect performances of workstations constituting the target distributed memory machines, the number of processors in each of these workstations, as well as bandwidths of links connecting the workstations (optionally).

All processes constituting the target program are divided into 2 groups - the special process named dispatcher playing the role of the computing space manager, and general processes named nodes playing the role of virtual processors of the computing space. The dispatcher works as a server accepting requests from virtual processors. The dispatcher does not belong to the computing space.

In the target program, every network or subnetwork of the source mpC program is represented by a set of nodes called region. So, at any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them is the responsibility of the dispatcher. The only exception is the pre-hired host-node representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

If a region represents a network, creation of the region involves the parent node, the dispatcher and all free nodes. The parent node sends creation request containing the necessary information about the network topology to the dispatcher. Based on this information and the information about the topology of the virtual parallel machine, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. Deallocation of network region involves all its members as well as the set of free nodes and the dispatcher.

If the region represents a subnetwork, its creation involves only members of the enclosing region. Deallocation of subnetwork region involves only its members.

The dispatcher keeps a queue of creation requests that cannot be satisfied immediately but can be served in the future. It implements some strategy of serving the requests aimed at minimization of the probability of occurring a deadlock. The dispatcher detects such a situation when the sum of the number of free nodes and the number of such hired nodes that could be released is less than the minimum number of free nodes required by a request in the queue. In this case, it terminates the program abnormally.

B.3. Supported systems

We tried to write all the components of the mpC programming environment in such a way to avoid any problem with its installation on any Unix system having C compiler supporting ANSI C. We have checked it for the following platforms:

- Sun workstations running Solaris 2.4/2.5, SunOS 4.1.3 with gcc versions 2.6.3, 2.7.0, 2.7.2 and SPARCworks Professional C 3.01;
- HP9000 workstations running HP-UX 9.07 with gcc version 2.7.2 and c89;
- DEC Alpha workstations running OSF1 with gcc version 2.7.2;
- PCs running Lunix 4.0 with gcc version 2.7.2.

We tried to write the mpC compiler in such a way to avoid any problem with compilation of generated code on any Unix system having C compiler supporting ANSI C. We have checked it for the platforms listed above.

We tried to write RTS in such a way to ensure its correct work for any implementation of MPI supporting full MPI 1.1 standard as an underlying comunication platform. We have checked it for LAM MPI versions 5.2, 6.0 (for the platforms listed above), for MPICH version 1.0.13 (for Sun workstations running Solaris and HP9000 workstations running HP-UX 9.07) and for HP MPI (for DEC Alpha workstations running OSF1).

The current version of the command-line user interface is written in such a way to work correctly for two implementations of MPI - LAM and MPICH. We have checked it for LAM versions 6.0 (for the

platforms listed above) and for MPICH version 1.0.13 (for Sun workstations running Solaris and HP9000 workstations running HP-UX 9.07).

B.4. mpC compiler

To call the mpC compiler one should type:

`mpcc [options] filename`

`mpcc` processes an input file through one or more of tree stages: preprocessing, analysis, and generating one or two C files. For preprocessing we use a standard preprocessor and recommend to use GNU `cpp`. Only one input file may be processed at once. The suffix `.mpc` is used for mpC source files, and the suffix `.c` is used for processed mpC files. `mpcc` puts output C files into the current directory.

B.4.1. Options

All options must be separated. For example, `-hetmacro` is quite different from `-het -macro`. All options different from described bellow are considered as options of the preprocessor.

`-E`

Stop after the preprocessing stage; do not run the compiler proper. The output is preprocessed source code, which is sent to standard output.

`-analyse`

Compiler provides parsing and semantical analysis. C files will not be generated.

`-k`*`mode`*

Selects one of four parser modes. *mode* may be SHORT, ANSI, LONG, and ALL. By default the SHORT mode is used. This mode allows to use only the short form of mpC keywords. The ANSI mode allows to use only ANSI C keywords. The LONG mode allows to use only the full form of mpC keywords. The ALL mode allows to use both forms of mpC keywords. For example, `net` and `mpc_net` are identifiers in the ANSI mode and mpC keywords in the ALL mode. `net` is an identifier in the LONG mode and an mpC keyword in the SHORT mode. Finally, `mpc_net` is an identifier in the SHORT mode and an mpC keyword in the LONG mode. Presence of these modes supports the compatibility with previously written C code.

`-macro`

Forbids to use some macros in generated C files. The macros contain parameterized code standing for long pieces of code. Code with macros is shorter but may be less undestandable.

`-out`

Directs output of `mpcc` to standard output instead of C file.

`-het`

Makes compiler produce two output C files. By default, for source mpC file *`name`*`.mpc` `mpcc` produces one output C file *`name`*`.c`. If `-het` is typed, then `mpcc` will produce two output files: *`name`*`_host.c` containing code for the virtual host-processor and *`name`*`_node.c` containing code for the rest of virtual processors. This option allows to isolate code with input/output operations to a single processor.

B.4.1. Pragmas

A `#pragma` directive of the form

`#pragma keywords_mode:`

is supported by `mpcc`. This pragma has the same affect on mpC keywords as option `-k` described above and allows one to use the same header files in C and mpC sources.

B.5. How to start up

By now, we dealt with local networks of workstations running UNIX (including PCs running LINUX) as a DMM. Any workstation that may be used as a working place of the user is called a host workstation. It is intended that the user may start running mpC applications only from host workstations.

To start working with the mpC environment, the user must have it installed on each of workstations constituting his DMM.

Then the user should become an authorised user with the same name on each of workstations constituting the DMM.

Then the user should make sure that on each of these workstations in his home directory file `.rhosts` exists and contains names of all workstations constituting the DMM.

Then on each of these workstations the user should modify corresponding files (for example, `.cshrc` if he uses C shell) in his home directory to determine environmental variables WHICHMPI, MPIDIR, MPCHOME, MPCTOPO, and MPCLOAD of his shell.

Notes:

- Sometimes one needs modify different files for local and remote invocation of shell. For example, for PC running Linux 4.0 the user should modify files `.bashrc` and `bash_profile` if he uses Bourne shell.
- When using LAM, it may be needed to determine environmental variable TROLLIUSHOME setting it to the same value as MPIDIR.
- When using MPICH, environmental variable MPCHOME must be set to the same value on all workstations constituting the DMM. To ensure it, the user may need to use the Unix `ln` command to make necessary hard or soft links.
- When using MPICH, environmental variable MPCLOAD must be set to the same value on all workstations constituting the DMM. To ensure it, the user may need to use the Unix `ln` command to make necessary hard or soft links.
- When using MPICH, the user should make sure that he has write access to directory `$MPIDIR/bin/machines` (equally, `$MPIDIR/util/machines`) on each of host workstations.

Then on each workstation the user should create his own directories $MPCTOPO, $MPCTOPO/log, and $MPCLOAD. No two workstations or users can share these directories. The user should make sure that he has write access to these directories.

Then on each workstation the user should modify corresponding files in his home directory to add directories $MPIDIR/bin, $MPIDIR/lib, $MPCHOME/bin, $MPCHOME/lib and $MPCLOAD to his PATH. To avoid name conflicts, make directory $MPCLOAD first in the search path.

In addition, the user should add directories $MPIDIR/lib and $MPCHOME/lib to his ld path (by changing LD_LIBRARY_PATH for Solaris, LPATH for HP-UX and so on).

B.6. Virtual parallel machine

The next step is the description of the virtual parallel machine (VPM) which will execute mpC applications. The description is provided by a manually-written VPM description file which should be placed to the $MPCTOPO directory. The name of this file is just considered as a name of the described VPM.

A VPM description file consists of lines of two kinds. Lines starting with symbol '#' are treated as comments. All other lines should be of the following format:

name number_of_processes

where *name* is the name of the corresponding workstation as it appears in the system `/etc/hosts` file, and *number_of_processes* is the number of processes to run on the workstation. The host workstation must go first in the file. The virtual host-processor will be mapped to a process running on this workstation.

For example, the following file describes VPM which runs on DMM consisting of three workstations (`alpha`, `beta`, and `gamma`), five processes running on each workstation, and the host workstation is `alpha`:

```
# three workstation each running 5 processes
alpha 5
beta 5
```

```
gamma 5
```

The following example describes VPM with the same total number of the processes, but running on the single workstation alpha. It may be useful for debugging mpC applications:

```
# simple topology for debugging
alpha 15
```

The actual total number of running processes is greater then the number specified in the description file. A process for the dispatcher is added automatically and runs on the host workstation. The virtual host-processor is always placed on the host workstation.

B.7. Environmental variables

WHICHMPI

Currently, $WHICHMPI should be either LAM or MPICH dependent on the used MPI implementation. WHICHMPI should be set to the proper value on host workstations.

MPIDIR

$MPIDIR is a directory where MPI has been installed. MPIDIR should be set to the proper value on each workstation of DMM.

MPCHOME

$MPCHOME is a directory where the mpC programming environment has been installed. MPCHOME should be set to the proper value on each workstation of DMM.

Subdirectory $MPCHOME/bin holds all executables and scripts of the mpC programming environment.

Subdirectory $MPCHOME/h holds all specific mpC header files as well as header `mpc.h` containing declarations of the mpC library and embedded functions.

Subdirectory $MPCHOME/lib holds RTS object files `mpcrts.o` and `mpctopo.o`.

When using MPICH, the user should ensure MPCHOME to have the same value on all workstations of the DMM. If mpC has been installed in different directories on different workstations, you can use the Unix `ln` command to make necessary hard or soft links and ensure the property.

MPCLOAD

$MPCLOAD is a directory for C files, object files, libraries and executables related to user's applications. MPCLOAD should be set to a proper value on each workstation of DMM. No two workstations or users can share the directory. The user should have write access to the directory.

When using MPICH, the user should ensure MPCLOAD to have the same value on all workstations of the DMM. In particular, you can use the Unix ln command to make necessary hard or soft links and ensure the property.

MPCTOPO

$MPCTOPO is a directory for VPM description files as well as all topological files produced by the mpC programming environment. MPCTOPO should be set to a proper value on each workstation of DMM. The mpC programming environment saves a file specifying the current VPM in subdirectory

$MPCTOPO/log. No two workstations or users can share these directories. The user should have write access to these directories.

B.8. How to run mpC applications

To run an mpC application on a described virtual parallel machine, the user should proceed the following steps:

- create the necessary VPM by the `mpccreate` command. Immediately after that, the VPM is opened;
- if the necessary VPM has been created earlier, open it by the `mpcopen` command instead of its creation;
- put all `.c` and `.o` user's files, necessary to produce executable file, into the $MPCLOAD directory on the host workstation;
- broadcast all the files, necessary to produce executable, from the $MPCLOAD directory on the host workstation to $MPCLOAD directories on other workstations constituting the DMM by the `mpcbcast` command;
- create an executable file on each of workstations constituting the DMM by the `mpcload` command;
- run the executables by the `mpcrun` command.

Additionally,

- `mpctouch` displays status of the VPM and all its processes.
- `mpcclean` cleans VPM.
- `mpcclose` ends the work with the current VPM.
- `mpcmach` prints name of the current VPM.

`mpccreate` *name*

where name is the name of the VPM to create. The command uses the $MPCTOPO/*name* description file. The command creates VPM, i.e. produces all necessary files for it.

In particular, the `mpccreate` command creates the $MPCTOPO/*name*.topo file, containing a description of the topology of the created VPM and used by RTS in run time. Currently, the file consists of pairs of lines of the form:

`#` *<name_of_workstation>*
`s`*<number_of_processors>* `p`*<performance>* `n`*<number_of_processes>*

where *<name_of_workstation>* is the name of the corresponding workstation as it appears in the $MPCTOPO/*name* description file, *<number_of_processors>* is the number of physical processors in the workstation, *<performance>* is an integer number characterizing the performance of each of these physical processors and *<number_of_processes>* is the number of processes running on the workstation. We recommend to check out the file after creation of the VPM, since the detected topological characteristics can be rough enough if the background workload of the corresponding DMM was essential and uneven during the work of the `mpccreate` command.

Once created, the VPM is accessible to be opened by `mpcopen`. Note that `mpccreate` is expensive and executes a lot of computations and communications, so it may take a few minutes to create new VPM.

`mpcopen` *name*

where *name* is the name of the VPM to open. The VPM must be created earlier. After opening, the VPM is accessible for `mpcbcast`, `mpcload`, `mpcrun`, `mpcclean`, `mpctouch`, `mpcmach`, and `mpcclose`.

`mpcbcast [`*file1 file2 ...* `]`

The command broadcasts files listed from directory $MPCLOAD on user's workstation to directory $MPCLOAD on the rest of workstations constituting DMM. Only file names without paths must be typed.

`mpcload [-het] -o` *target* `[`*file1.c file2.c ...* `]`

`[file01.o file02.o ... ]  [file11.a file12.a ... ]`
`[options_to_all_nodes] [-host] [options_to_host_only]`

The `mpcload` command produces executable target from in directory $MPCLOAD on each of workstations constituting the DMM. Do not use a path in target.

The command produces the executable from `.c`, `.o` and `.a` files. The name of each of these files either uses no path or uses the full path. The first case means that the file is searched in directory $MPCLOAD.

Option `-het` must be used if workstations participating in the VPM are not binary compatible. The user may use the option even if all the workstations are binary compatible.

Option `-host` separates options necessary to all nodes and options necessary only for the virtual host-processor. In addition, if this option appears then:
target for the virtual host-processor is produced from fully-named files and shortly-named files, whose names are produced from names of shortly-named `.c`, `.o` and `.a` files as they are typed in the command line by addition `_host` to the end of name;
target for other nodes is produced from fully-named files and shortly-named files, whose names are produced from names of shortly-named `.c`, `.o` and `.a` files as the are typed in the command line by addition `_node` to the end of name.

`Options_to_all_nodes` and *`options_to_host_only`* are may be any proper C compiler options. Note, that if option `-c` is used, and hence target is an `.o` file, then option `-host` can not be used.

`mpcrun` *`target`* `[--` *`params`*`]`

The command runs mpC application *`target`* on the current VPM and passes parameters *`params`* to this application. Do not use a path in target.

`mpctouch [-p]`

The command checks status of the current VPM and displays it. The VPM may be ready or busy. If option -p is typed then the status of all nodes is displayed. Currently, the command makes sense only for LAM implementation.

`mpcclose`

Closes the current VPM.

`mpcclean`

The command cleans the current VPM and makes it ready to run new mpC application. The command should be used in case of abnormal termination of the previous command or mpC application. Currently, the command makes sense only for LAM implementation.

`mpcmach`

Prints the name of the current VPM.

`mpcdel` *`name`*

where *`name`* is the name of a VPM. The command deletes the VPM (that is, deletes all system files related to the VPM).

B.9. Debugging mpC applications recommendations

Debugging an mpC application isn't yet an easy task, but it is much simpler than debugging an arbitrary MPI application, because of absence of nondeterminism.

There are at least three levels of debugging.

At the top level we suggest to include in mpC code calls to `MPC_Global_barrier()` and `MPC_Barrier()` to split program execution into small debuggable portions. It may be helpful to use the `MPC_Prinf()` function to output node coordinates, and values of variables. But there are no guarantee you to see all `MPC_Printf` messages, because some errors make message-passing subsystem failed. It is also possible to use `printf`, but part of output done on remote computers will be lost. However, we strongly recommend to start debugging using a single workstation as DMM. All error messages include either position in the mpC source file or `0,0`, if the error takes place in the dispatcher process.

The middle level includes including `printf`'s and barriers in C code generated by `mpcc`. It is a bit more sophisticated than previous approach, because the user needs to understand the logic of the generated C code and RTS kernel calls. If the user sets environmental variable MPCTRACEMAPFILE to an absolute name of file, then in the corresponding file he will obtain a table, where each pair of lines contains info about network allocation. Sign "+" stands for previously allocated process, sign "-" stands for currently unemployed one, and "p" - for the parent node of the network. In the second line user can see node ranks in the created network.

At the low level of debugging, the user may turn kernel tracing on and use MPI utilities such as `state` and `mpitask` (LAM 6.0). To turn tracing on, the user needs to close the current virtual parallel machine, then set the MPC_DEBUG environmental variable to 1 or 2 and reopen the machine.

To debug an application, which hangs without error messages, first call `mpitask` to find processes which do not respond. In many cases they "die" due a simply "C error", such as an uninitialized variable, dividing by zero and so on. The trace is useful for finding the point in the code where the disaster occurs. When all processes are alive, `mpitask` shows operations, where the processes are blocked.

B.10. Example of mpC session

Let the virtual parallel machine to run the application `gal-buf.mpc` has been already opened. To produce two target C files - the first for the virtual host processor executing code that includes calls to Xlib displaying data in the graphical form, and the second for the rest of virtual processors not involved in graphical representing data, one can type:

```
mpcc -I/usr/openwin/include -het gal-buf.mpc
```

Note. Use the absolute application name if `gal-buf.mpc` is not in the current directory. Use directory other then `/usr/openwin/include` if necessary (that is, use the directory where X Windows system holds its include files on the host workstation).

The above command will produce files `gal-buf_host.c` and `gal-buf_node.c` in the current directory. To make these files accessible to the mpC programming environment, one should copy them into the $MPCLOAD directory:

```
cp gal-buf_host.c gal-buf_node.c $MPCLOAD
```

To broadcast these files from the host workstation to all workstations constituting the distributed memory machine, one should type:

```
mpcbcast gal-buf_host.c gal-buf_node.c
```

To produce executable gal-buf on each workstation of the distributed memory machine, one can type:

```
mpcload -het -o gal-buf gal-buf.c -lm -host -L/usr/openwin/lib -lX
```

Note. Use a directory other then `/usr/openwin/lib` if necessary (that is, use the directory where X Windows system holds its libraries on the host workstation). Use an option other then `-lX` if necessary (that is, use the proper name for the X library; it may be `-lX11` or something else).

Finally, to run the application, one can type:

```
mpcrun gal-buf -- input_file
```

where file *`input_file`* contains input data for the application.

Note. Use the absolute name of the input file if it is placed in a directory other then the directory which was a current directory when you open your virtual parallel machine.

Appendix C

C.1. Cholesky factorization in mpC and in ScaLAPACK

Here, we compare mpC and ScaLAPACK to demonstrate that unlike the traditional technology, supported by ScaLAPACK and HPF, the technology, supported by mpC, allows one to develop parallel applications efficiently portable among heterogeneous networks. Currently, ScaLAPACK is one of the most advanced and famous free packages built on top of MPI. The design philosophy of the ScaLAPACK library is typical (in particular, the same as that of HPF) and based on a homogeneous multiprocessor with very fast communications among processor nodes as a model of the parallel machine executing applications.

ScaLAPACK is a package for solving linear algebra problems on massively parallel, distributed memory, concurrent computers. It has been designed as a message-passing version of LAPACK. It is based on two additional packages - Basic Linear Algebra Communication Subprograms (BLACS), and Parallel BLAS (PBLAS). The BLACS provides communication between processes on a logical two-dimensional process grid for linear algebra purposes and encapsulates a particular communication platform (in particular, MPI). The PBLAS is an extended subset of the BLAS for distributed memory computers and operates on matrices distributed according to a block cyclic data distribution scheme. It calls BLAS for computations and calls BLACS for communications.

For the comparison we selected a high-level ScaLAPACK routine implementing parallel Cholesky factorization of a real symmetric positive definite matrix. Cholesky factorization is an extremely important computation, arising in a variety of scientific and engineering applications. It is a well-known challenge for efficient and scalable parallel implementation because of large volumes of interprocessor communications.

Cholesky factorization factors an nxn, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$. One can partition the nxn matrices A, L and LL^T and write the system as

$$\begin{matrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{matrix} = \begin{matrix} L_{11} & 0 \\ L_{21} & L_{22} \end{matrix} \times \begin{matrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{matrix}$$

where blocks A_{11} and L_{11} are nb1* nb1, A_{21}, L_{21} are (n-nb1)*nb1, A_{22}, L_{22} are (n-nb1)*(n-nb1). L_{11} and L_{22} is lower triangular, nb1 is size of the first block. If to assume that L_{11}, the lower triangular Cholesky factor of A_{11}, is known, one can rearrange the block equations

$$L_{21} \leftarrow A_{21} (L_{11}^T)^{-1}$$

$$A_{22}^{\sim} \leftarrow A_{22} - L_{21} L_{21}^T = L_{22} L_{22}^T$$

The factorization can be done by recursively applying the step outlined above to the updated matrix $A_{22}^{\sim}$. The parallel implementation of the corresponding ScaLAPACK routine is based on the above scheme and a block cyclic distribution of the matrix A over a PxQ process grid with a block size of nbxnb. The routine assumes that the lower (upper) triangular portion of A is stored in the lower (upper) triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A (here and henceforth when speaking of an array we mean a Fortran array, that is, that column elements of an 2-D array are allocated contiguously).

Our mpC Cholesky factorization routine implements almost the same algorithm [18] that is implemented by the ScaLAPACK one when P=1 and the lower triangular portion of A is used to compute L. Namely, to compute above steps it involves the following operations:

- process `Pr`, which has L_{11}, L_{21}, calls LAPACK function `dpotf2` to compute Cholesky factor L_{11} and sets a flag if A_{11} is not positive define;
- process `Pr` calls BLAS function `dtrsm` to compute Cholesky factor L_{21} if A_{11} is positive define;
- process `Pr` broadcasts the column panel, L_{11} and L_{21}, as well as the flag to all other processes and stops the computation if A_{11} is not positive define;
- all processes update matrix A_{22} in parallel, that involves calls to BLAS functions `dsyrk` and dgemm by each process to update its local portions of the matrix A_{22}.

The main difference between the mpC and ScaLAPACK routines concludes in data distribution. In fact, in our case the ScaLAPACK routine divides the matrix A into a number of column panels with just the same width nb and distributes them cyclically over Q processes where Q, nb are input parameters of the routine.

The mpC routine divides the matrix A into k column panels with different widths n1, n2,..., nk, where n1+n2+...+ nk=n, and distributes (in general, non-cyclic) them over Q processes. The routine performs this partition in run time based on the following data:

- the total number of actual processors and their performances both returned by the function `MPC_Processors_static_info`;
- the communication speed of an underlying network returned by the function `MPC_Links_static_info`;
- the size n of the matrix A.

The values of Q, k and n1, n2,..., nk as well as the mapping of the column panels onto the Q processes are computed in such a way to ensure the most efficient running of the application on the underlying heterogeneous network. Since our model of a heterogeneous network takes into account not only the number of processors but also their performances and speed of communications among them, the optimal data distribution may be not as regular as ScaLAPACK data distribution. In particular, the number Q of processes, taking part in the computations, may be less that the total number of processors. Taking into account the speed of communications may cause non-cyclic distribution of column panels. To balance the load of processors of different performances, the widths of column panels should be different. In addition, to achieve the balance between computations and communications, widths of column panels will grow from left to right.

C.2. Experimental results

We compared the running time of our mpC program and its ScaLAPACK counterpart. We used three SPARCstations 5 (hostnames `gamma`, `beta`, and `delta`), and SPARCclassic (`omega`) with relative performances 160, 160, 160, and 77 correspondingly connected via 10Mbits Ethernet. We used several distributed memory machines (DMMs): `gbo`, constituting of `gamma`, `beta`, and `omega`, `gbd`, consisting of `gamma`, `beta`, and `delta`, and so on. Note, that topological characteristics of these DMMs (processor performances and speeds of communication) are detected automatically by a command of the mpC programming environment.

We used MPICH version 1.0.13 as a particular communication platform, GNU C compiler with optimization option -O2, and GNU fortran compiler with optimization option -O4. As a base of the comparison we used the running time of the LAPACK function dpotf2 implementing a sequential Cholesky factorization. We started one process per workstation for both programs and used `nb`=5 (providing the best performance) for the ScaLAPACK one.

Table 1 gives speedups computed relative to the LAPACK routine running on gamma. One can see that the mpC program and the ScaLAPACK routine take the same time when running on homogeneous DMMs `gb` and `gbd`. If to enhance these machines with low-performance omega, then the mpC program allows to utilize the parallel potential of performance-heterogeneous `gbo` and `gbdo`, speeding up the Cholesky factorization. At the same time, its ScaLAPACK counterpart does not allow to do it slowing down the Cholesky factorization. Note, that all above DMMs, consisting of Sun workstations, are characterized by the same communication speed, high enough the mpC program to involve all workstations in Cholesky factorization.

Table 1: Speedups computed relative to the LAPACK programm running on gamma (ScaL - ScaLAPACK)

n	gb		gbo		gbd		gbdo	
	mpC	ScaL	mpC	ScaL	mpC	ScaL	mpC	ScaL
300	1.03	1.03	1.15	0.85	1.25	1.27	1.25	1.02
400	1.13	1.10	1.29	0.98	1.47	1.42	1.50	1.17
500	1.18	1.17	1.38	1.04	1.56	1.54	1.64	1.29
600	1.27	1.24	1.48	1.09	1.69	1.65	1.76	1.36
700	1.29	1.26	1.53	1.13	1.75	1.75	1.85	1.41
800	1.33	1.32	1.57	1.12	1.82	1.84	1.90	1.49